1-a

Concept	Explanation
Tensors	Multi-dimensional arrays used to store data.
Neural Network	A simple model with 2 layers to learn patterns.
Loss Function	MSE loss measures prediction error.
Optimizer	SGD optimizer adjusts model weights.
Training Loop	Model learns by updating parameters.
Evaluation	Model predicts unseen data after training.

1. Import Libraries

python
Copy code

```
import torch
import torch.nn as nn
import torch.optim as optim
```

Purpose:
Imports essential libraries for PyTorch.

- `torch`: Main library for tensor computation.
- `nn`: Contains neural network building blocks like layers and activation functions.
- `optim`: Contains optimization algorithms like SGD, Adam, etc.

---

2. Tensor Creation

python
Copy code

```
x = torch.tensor([1.0, 2.0, 3.0])  # 1D tensor (vector)
y = torch.tensor([4.0, 5.0, 6.0])  # 1D tensor (vector)

print(f"x: {x}")
print(f"y: {y}")
```

Purpose:
Creates two 1D tensors `x` and `y` containing 3 elements each.

- `x = [1.0, 2.0, 3.0]`
- `y = [4.0, 5.0, 6.0]`
  These are printed for reference.

---

3. Basic Tensor Operations

python
Copy code
```
z = x + y
print(f"z (x + y): {z}")

z = x * y
print(f"z (x * y): {z}")
```

Purpose:
Performs arithmetic operations on the tensors.

- **Addition**: `z = x + y = [1+4, 2+5, 3+6] = [5, 7, 9]`
- **Multiplication**: `z = x * y = [1*4, 2*5, 3*6] = [4, 10, 18]`

---

4. Neural Network Model Definition

python
Copy code
```
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(3, 5)  # Fully connected layer (input=3, output=5)
        self.fc2 = nn.Linear(5, 1)  # Fully connected layer (input=5, output=1)
        self.relu = nn.ReLU()      # ReLU activation function

    def forward(self, x):
        x = self.relu(self.fc1(x))  # Pass input through first layer, apply
ReLU
        x = self.fc2(x)             # Pass through second layer
        return x
```

Purpose:
Defines a simple neural network with 2 fully connected (FC) layers.

- **Input to Layer 1**: 3-dimensional input, output of size 5.
- **Input to Layer 2**: 5-dimensional input, output of size 1.
- **Activation**: ReLU (Rectified Linear Unit) is applied after the first layer.
- **Forward Pass**: Describes how data flows through the model.

---

5. Instantiate the Model

python
Copy code
```
model = SimpleNN()
```

Purpose:
Creates an instance of the `SimpleNN` model, initializing its layers and parameters.

---

6. Define Loss Function and Optimizer

python
Copy code

```
criterion = nn.MSELoss()  # Mean Squared Error loss
optimizer = optim.SGD(model.parameters(), lr=0.01)  # Stochastic Gradient
Descent optimizer
```

Purpose:

- **Loss Function**: Mean Squared Error (MSE) is used to compute the difference between predicted and actual values.
- **Optimizer**: SGD (Stochastic Gradient Descent) updates the model parameters during backpropagation. The learning rate `lr=0.01` controls the step size.

---

7. Dummy Input and Target Data

python
Copy code

```
input_data = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
target_data = torch.tensor([[10.0], [20.0], [30.0]])
```

Purpose:
Creates sample **input data** and **target labels** for training.

- `input_data` has 3 samples, each with 3 features.
- `target_data` has the expected output for each of the 3 samples.

---

8. Training Loop

python
Copy code

```
epochs = 100
for epoch in range(epochs):
    model.train()  # Set the model to training mode
    optimizer.zero_grad()  # Clear gradients from previous step

    # Forward Pass
    outputs = model(input_data)

    # Calculate Loss
    loss = criterion(outputs, target_data)

    # Backward Pass (Backpropagation)
    loss.backward()

    # Update Model Parameters
    optimizer.step()

    # Print Loss every 10 epochs
```

```
    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/100], Loss: {loss.item():.4f}")
```

Purpose:
This is the main training loop where the model learns from data.

1. **Epochs**: Trains for 100 iterations.
2. **Training Mode**: The model is set to training mode using `model.train()`.
3. **Gradient Zeroing**: Clears previous gradients.
4. **Forward Pass**: Sends input data through the model to get predictions.
5. **Loss Calculation**: Compares the predictions with target labels using MSE.
6. **Backward Pass**: Computes the gradients via backpropagation.
7. **Parameter Update**: Updates the model's parameters using the optimizer.
8. **Loss Logging**: Prints the loss every 10 epochs to track training progress.

---

9. Making Predictions

python
Copy code

```
model.eval()  # Set model to evaluation mode
with torch.no_grad():  # No gradients are needed for inference
    predictions = model(input_data)
    print("\nPredictions:")
    print(predictions)
```

Purpose:
Once training is complete, the model is tested on the same `input_data`.

1. **Evaluation Mode**: `model.eval()` disables dropout and batch normalization.
2. **No Gradients**: Uses `torch.no_grad()` to avoid tracking gradients for efficiency.
3. **Prediction**: Uses the trained model to predict outputs for the `input_data`.
4. **Print Results**: Prints the predictions.

1-b

Concept	Category	Description
Word2Vec Embeddings	**NLP / Embeddings**	Converts words into fixed-size vectors capturing semantic meaning using Gensim's pre-trained Word2Vec model.
Text Preprocessing	**NLP / Data Preparation**	Cleans text by removing non-alphabetic characters and tokenizing it into words.
Embedding Averaging	**NLP / Embeddings**	Averages word embeddings for all words in a sentence to create a single vector representing the sentence.

Train-Test Split	**Data Preparation**	Splits data into training and validation sets to evaluate model performance on unseen data.
DataLoader	**PyTorch / Data Handling**	Loads training and validation data in mini-batches to improve training efficiency.
MLP (Multi-Layer Perceptron)	**Neural Network Architecture**	A simple neural network with an input layer, one hidden layer, and an output layer for binary classification.
ReLU Activation	**Activation Function**	Adds non-linearity to the network, allowing it to model complex relationships in data.
Softmax Function	**Activation Function**	Converts output logits into probabilities for binary classification.
CrossEntropyLoss	**Loss Function**	Measures the difference between predicted probabilities and true labels for classification tasks.
Adam Optimizer	**Optimization Algorithm**	An adaptive learning rate optimization algorithm used to update model weights.
Backpropagation	**Training Concept**	A method to compute gradients of the loss with respect to model weights to update weights via gradient descent.
Batch Training	**Training Concept**	Processes mini-batches of data at a time instead of the entire dataset, leading to faster and more stable training.
Model State Saving	**Model Persistence**	Saves the model's learned parameters (weights) for later use or transfer learning.
Evaluation Metrics	**Model Evaluation**	Metrics like accuracy, precision, recall, and F1-score are used to measure the model's performance.
Classification Report	**Model Evaluation**	Provides detailed evaluation metrics (precision, recall, F1-score) for each class.
Prediction on New Data	**Model Inference**	Uses the trained model to classify new reviews as Positive or Negative.
Dynamic Device Allocation (GPU/CPU)	**Hardware Utilization**	Dynamically switches to GPU if available, otherwise uses CPU for computation.
Data Batching	**Training Efficiency**	Uses mini-batches of data instead of the whole dataset to improve computational efficiency.
Gensim API	**Pre-Trained Embeddings**	Gensim's API allows for the direct loading of pre-trained Word2Vec models.
Tensor Conversion	**Data Handling**	Converts NumPy arrays to PyTorch tensors for GPU acceleration and neural network training.

Greedy Tokenization	**Text Tokenization**	Splits text into words using simple whitespace-based tokenization.
Sentiment Classification	**Machine Learning Task**	Classifies text as "Positive" or "Negative" using a neural network.

script implements a sentiment analysis model using PyTorch. Here's a concise explanation of its key components:

1. **Imports and Device Setup**:
    - Required libraries like PyTorch, NumPy, Gensim, and Scikit-learn are imported.
    - The script checks if a GPU is available and sets it as the computation device.
2. **Word2Vec Embeddings**:
    - The Gensim Word2Vec pre-trained embeddings are loaded.
    - A function converts input text into Word2Vec embeddings by averaging the embeddings of individual words.
3. **Data Preparation**:
    - Text features (`X`) and sentiment labels (`y`) are prepared.
    - The data is split into training and validation sets.
    - PyTorch `DataLoader` is used to create batches for training and validation.
4. **MLP Model Definition**:
    - A simple **Multi-Layer Perceptron (MLP)** with one hidden layer is defined.
    - It uses ReLU activation and a Softmax output for binary classification (positive vs. negative sentiment).
5. **Loss Function & Optimizer**:
    - Cross-entropy loss and Adam optimizer are used for training.
6. **Training**:
    - The model is trained for 10 epochs.
    - Loss is calculated for both training and validation sets after each epoch.
    - The training and validation loss are plotted.
7. **Saving & Loading Model**:
    - The trained model's parameters are saved to a file.
    - The saved model is loaded to make predictions on new data.
8. **Prediction on Sample Reviews**:
    - New reviews are converted to embeddings, passed through the model, and classified as Positive or Negative.
9. **Evaluation**:
    - Model predictions are compared to the true labels on the validation set.
    - Accuracy, precision, recall, F1-score, and a classification report are printed.

This end-to-end pipeline for sentiment classification using a pre-trained Word2Vec model, an MLP classifier, and performance evaluation on test data.